Lineal Feature Extraction by Parallel Stick Growing

G.C. Hunt and R.C. Nelson

Technical Report 625
June 1996

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1996	technical report

4. TITLE AND SUBTITLE

Lineal Feature Extraction by Parallel Stick Growing

5. FUNDING NUMBERS

ONR N00014-93-I-0221

6. AUTHOR(S)

Galen C. Hunt and Randal C. Nelson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)

Office of Naval Research
Information Systems
Arlington VA 22217

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TR 625

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

line detection; feature extraction; irregular parallel processing
lock preemption; deadlock prevention

15. NUMBER OF PAGES

12 pages

16. PRICE CODE

free to sponsors; else $2.00

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

Lineal Feature Extraction by Parallel Stick Growing

Galen C. Hunt * and Randal C. Nelson **

Department of Computer Science
University of Rochester, Rochester, NY 14627, USA
{gchunt,nelson}@@cs.rochester.edu

Abstract. Finding lineal features in an image is an important step in many object recognition and scene analysis procedures. Previous feature extraction algorithms exhibit poor parallel performance because features often extend across large areas of the data set. This paper describes a parallel method for extracting lineal features based on an earlier sequential algorithm, stick growing. The new method produces results qualitatively similar to the sequential method.
Experimental results show a significant parallel processing speed-up attributable to three key features of the method: a large numbers of lock preemptible search jobs, a random priority assignment to source search regions, and an aggressive deadlock detection and resolution algorithm. This paper also describes a portable generalized thread model. The model supports a light-weight job abstraction that greatly simplifies parallel vision programming.

1 Introduction

Finding lineal features in an image is an important step in many object recognition and scene analysis procedures. It is also a time-consuming one. Even with modern workstations, finding all of the lines in a single image can take on the order of tens of seconds. Given the need for speed, lineal feature extraction is an obvious candidate for parallel processing. However, unlike many image processing operations, such as convolution, and more general local area operations such as relaxation and morphological transforms, higher level feature extraction computations do not have a regular structure that can be easily exploited by automatic parallelization techniques. Extended feature detection operations in general, and lineal feature extraction algorithms in particular tend to make disproportionate access to image data elements in the features (e.g., along the lines). It is difficult to predict the computational pattern in advance because feature locations are known only after the computation has finished. Extended feature extraction processes in vision represent an important class of irregularly structured problems for which efficient parallel algorithms are needed.

We present here a parallel method for lineal feature extraction suitable for coarse-grain implementation. The method is a parallel adaptation of the line-segment finding procedure described in [10]. Although we implement an algorithm for the extraction of

* Galen Hunt was supported by a research fellowship from Microsoft Corporation.
** This work was supported in part by NSF IIP grant no. CDA-94-01142, and ONR research grant no. N00014-93-I-0221.

a specific kind of extended feature (line segments), we think that the general parallel approach can be applied to a wide variety of extended feature extraction processes. In particular, the original serial algorithm has been since modified to extract curvilinear boundaries. Carrying over the parallel implementation in this case is trivial.

The layout of the paper is as follows. The remainder of this section gives a brief introduction to lineal-feature extraction methods and previous parallel work. Section 2 briefly describes Nelson's stick-growing method for segment extraction. Section 3 describes our programming model and the parallel stick growing method. In Section 4 we present experimental results demonstrating the strong scalability of the new method. Concluding remarks and future work can be found in Section 5.

1.1 Lineal Feature Extraction

There have been several approaches to extracting lineal primitives. The most widely used involves edgel linking and segmentation. The basic idea is to find local edge pixels using some low-level process, link them into contours on the basis of proximity and orientation, and then segment the contours into relatively straight pieces, again using any of several processes. The classic example of this approach is the Nevatia Babu line detector [11]. Other examples include work by Zhou et al. [15] and Nalwa and Pauchon [9]. Difficulties with the linking approach are basically due to its locality, and include unreliability of the low-level edge finder, instability of segmentation in the presence of bumps or many low-level edges, and difficulty hooking up long features if the data are sparse. Some of these problems can be ameliorated using multi-resolution representations e.g., [4], and grouping techniques [7].

A second method of line detection is based on the Hough transform [2]. Here local edges vote for all possible lines they are consistent with, and the votes are tallied up later to determine what lines are actually present. The main problems with this approach are complexity, coarse resolution, and lack of locality. The method is also expensive to implement, particularly if high resolution is desired, because every edgel must vote for all the lines it is consistent with. This problem is sometimes addressed in a post-processing, verification phase. The method also has trouble finding short segments in busy images. Princen et al. [13] address some of these problems using a hierarchical grouping process in conjunction with a local Hough transform.

A third method of lineal feature detection due to Burns et al. [1] utilizes the gradient direction to partition the image into a set of support regions, each of which will presumably be associated with a single feature. A least-squares fitting procedure is then used to fit a line segment to each region. This method can detect low-contrast features, but the segmentation can be unstable. Also features can rather easily be broken up by local perturbations.

Finally, there are statistical approaches. For example, Mansouri et al. [8] propose a hypothesize and test algorithm to find line segments of a given length by hypothesizing their existence based on local information, and attempting to verify that hypothesis statistically on the basis of a digital model of an ideal segment edge.

The method we implement here, is closest to the linking approaches, but uses a growth rule based upon a non-linear energy minimization scheme over a broad region of support in order to avoid the locality problems of traditional linking methods.

1.2 Previous Parallel Work

Our parallel method is significant because we parallelize an irregular algorithm with robust extraction capabilities. Prior works have parallelized regular algorithms with limited extraction capabilities.

Little [6] describes a feature extraction implementation on the massively parallel Connection Machine with 64K processors. Each pixel in the image is assigned to a separate virtual processor. Edgel detection is performed by convolving the image with a Gaussian operator. Each pixel is linked with any existing neighboring pixel. The pixels in a contour are labeled by an iterative distance-doubling algorithm. Each processor is assigned a fixed unique integer identifier. At every step of the algorithm each processor exchanges with its linked neighbors the maximum identifier known to belong to the contour. Little's algorithm provides edgel detection and linking, but does not segment contours into extended features.

Lin et al. [5] describe a parallel algorithm in which each processor performs a variant of the Nevatia-Babu [11] algorithm on a small region of the image. If the pixels of a contour cross a region boundary, the lines from each region are joined using a linear approximation algorithm credited to Williams [14]. They report a weighted speedup of 78 using 4096 processors on a MasPar MP-2 and a speedup of 307 on a 512 processor CM-5 [12].

Gerogiannus and Orphanoudakis [3] describe a parallel implementation of the Hough transform [2]. The image is broken into discrete regions with each region assigned to a processor. Each processor calculates the votes by the edges within its region for lines in the Hough space. The votes are then summed across processors. Using an iPCS/2, they report speedups of 4 for 64 processors. An implementation of the Hough transform using lock-free increment instructions for voting could achieve significantly higher speedup.

2 Stick Growing

Our parallel algorithm is based on a sequential line finding procedure described in [10]. The essence of the method is to define a metric that assigns a score to any possible line segment, based on the underlying image data, and repeatedly extract the best segment from the image. The practical problems are first, to design an appropriate matching measure, and second, to make the method efficient since it is clearly impractical to look through all possible segments multiple times. The efficiency problem can potentially be dealt with using any of several approximate maximization techniques. In this case, the problem is well enough behaved that a hill climbing method is effective.

The issue of designing a matching criterion for mapping lineal features to line segments is a bit subtle. The main difficulty is that, while a line segment is well defined mathematically, the notion of a lineal feature is a subjective one and must be dealt with as such. Intuitively, a lineal feature consists of a straight part, and two ends. Hence we define a matching criterion that includes explicit representations for the straight section of the feature, and the end stops. These end stops turn out to be extremely important in achieving good performance. We call the combined representation a *stick*.

The matching criterion is applied to the image as follows. The gradient magnitude and direction are determined by local convolution operators. To compute the match-score for a particular stick, the magnitude image is correlated with three templates at the appropriate positions and orientations. One of these represents a straight segment, the other two end stop patterns. The straight segment is a Gaussian whose central profile has been linearly extended, and the end stop patterns are differences of Gaussians with centers separated by two standard deviations. The correlation is computed using only points whose gradient direction is consistent with the direction of the segment. The match score is computed by adding the straight correlation value to any positive response from the end stop measures. Negative values from the end-stop templates are set to zero. This non-linearity prevents a sudden brightening in the line from inhibiting the growth of a stick.

Sticks are fitted to lineal features by first finding a high-gradient starting point. These starting points are edgels, determined by applying non-maximum suppression in the direction of the gradient. Starting with a short initial stick aligned perpendicular to the local gradient and centered at the starting point, a hill climbing procedure is performed, varying the centerpoint, length, and orientation of the stick incrementally to maximize the match score.

A few additional practical details are involved. Since it is inefficient to compute the entire straight correlation at each step for longer sticks, the full match value is computed only for sticks up to a certain length (about 14 pixels in the current implementation). This constitutes a seed segment. Beyond that point, the effect of extension at both ends is explored by probing out from a base point (initially the center point of the seed) with extension templates with a restriction that the orientation can change only slightly from that of the seed. Should extension be indicated, then a new basepoint is selected from among the three adjacent pixels that would increase the length of the stick by finding the best match among the nine permitted basepoint/angle combinations. When a local maximum is finally reached, if the final stick has a length exceeding a selected threshold, the points contributing to the final score are marked, and eliminated from contributing to other segment scores.

In order to find multiple segments, the image is broken up into neighborhoods, and sticks are grown starting at the top N locations in each neighborhood. A new starting location is not selected until completion of the growth phase of the previous stick, since some previously attractive start locations may be subsumed by the new feature. A stick can grow out of its original neighborhood, which can have the effect of eliminating some start-point candidates in others it passes through.

In summary, the stick growing algorithm operates as follows:

1. Preprocess image and extract gradients and local edge points.
2. Break image into small 32x32 pixel regions.
3. For each region:
   (a) Find edgel point with highest gradient.
   (b) Grow segment up to 14 pixels long perpendicular to gradient using template matching.
   (c) If segment is longer than 14 pixels, try to grow the tips as follows:
      i. Start each tip as one half of the segment.

ii. Perform template match (nonlinear) for possible tip extensions.
iii. As long as extensions correlate at least as well as original line segment extend the tip.

(d) Record the line segment (stick), erase gradient and edge pixels used in stick construction and look for another segment in the same region.

3 Parallel Stick Growing

Creating a good parallel algorithm requires a sound programming model and a clear insight into where potential parallelizations exist within a program. In this section we describe first, the programming model used for creating our parallel stick growing method. We then describe the parallelization possible within stick growing and our methods for exploiting it.

3.1 Programming Model

Our parallel stick growing method uses a generalized thread model (GTM). GTM abstracts a shared-memory system providing the creation and destruction of parallel threads and re-entrant mutexes. GTM is implemented as a small set of abstract C++ classes defining a system, threads and mutexes. Concrete classes for each operating system inherit and expand these abstract classes. GTM implementations exist for Sun Solaris, SGI IRIX, Digital Unix (OSF/1) and the Win32 API (Windows 95 and Windows NT). Porting GTM to a new system requires modifying only a single file which is normally on the order of 200 lines of well-documented C++ code.

Although threads are normally considered light-weight, they can be quite expensive when compared to process granularities in most vision algorithms which can be as small as a few hundred CPU cycles. For instance, during preprocessing stick growing uses a 3x3 pixel averaging filter. Theoretically, the optimal granularity would be a set of 9 input pixels and single output pixel. In reality, false sharing effects within cache lines enlarges the smallest practical granularity to something on the order of one hundred input pixels, which corresponds to several thousand CPU cycles. Cache-line effects are small compared to the cost of creating and destroying processor threads. On our SGI Challenge, thread start-up times are usually on the order of 250ms. Given a 100MHz clock cycle, thread creation takes approximately 25 million CPU cycles. Thread creation times dwarf most vision algorithm granularities.

In order to reduce the smallest possible CPU scheduling granularity, we introduce into our programming model the concept of *jobs*. Abstractly, a *job* is the smallest indivisible unit of work available in an algorithm. In practice, the size of a job is limited by cache-line effects. Jobs normally have a minimal size of a few tens-of-thousands of CPU cycles. We implement jobs as concrete classes derived from an abstract C++ class. All state for a job must be contained within its class instance. In addition to state specific to its work, a job also has a fixed priority assigned at the time of its creation. Runnable jobs are placed in a priority heap. Worker threads, one per processor, remove a single job at a time from the queue. Optimal load balance is guaranteed for any specific job granularity. The worker thread executes the job by calling its `run` member function.

The job runs non-preemptively until it either finishes or returns itself to the queue. Because all of a job's state is maintained within its class instance, worker threads use a single stack. Another positive benefit is that job don't need any kind of locking of their private state because they can never be preempted. Context switching between jobs consists of nothing more than selecting a job pointer from the heap and calling its `run` member function. Job creation requires only a single, typically small, dynamic memory allocation for the class instance member functions and a function call to add the job to the queue.

The combination of a highly portable generalized thread model (GTM) and an extremely light-weight job abstraction, provides a powerful programming model for vision algorithms. With GTM, our code is readily runnable on a large number of machines. Jobs strengthen the model by creating a unit of processing with granularity close to that exhibited by most vision problems.

3.2 Parallelization

Parallelism in stick growing exists in two forms, the first is data parallelism in the initialization step. Stick growing correlates line segments using a large set of templates. We reduce algorithm start-up time by creating templates in parallel. Another costly, but parallelizable, task is image preprocessing. The largest fraction of image preprocessing in stick growing consists of a number of 3x3 averaging and gradient calculation operations. We parallelize preprocessing by creating distinct jobs which operate on horizontal strips just 8 pixels wide.

The vast majority of processing time in stick growing is spent finding and growing line segments. The obvious units for parallelization are the 32x32 search regions. We create one job for each search region. The job's task is to extract the lines starting within its region. Each search job is assigned a unique fixed priority. As will be shown in section 4, assigning priorities randomly increases the chance that temporally concurrent jobs are spatially distant. Spatially distant reduces the chance that two jobs will contend for the same region.

Because lines almost always extend beyond a single region, there exists a need to synchronize job access to image regions. The stick search and grow algorithm is implemented using a roll-back enable transaction mechanism. A job searches for a line segment then grows it using a sequence of states. All state information about a line is maintained in such a way that line growing can be suspended or rolled back to the start configuration at any time when a search job attempts to enter a new region.

When a job has found a line, it commits by erasing the gradient and edge information subsumed by that line. Jobs must not erase data being used by another job. Synchronization becomes even more important in light of the fact that lines always have two ends and often cross each other. A job must hold a lock on a region before it can touch any data within the region. A job holds locks to all regions it has touched while growing a line segment until it either commits or is forced to roll back.

Before entering a new region, a job attempts to grab the lock for the region. If the lock is not held by another job, the job continues growing its current stick. If the lock is held by another job, the former job suspends itself. The suspension releases the worker thread to the next job available on the work queue.

Because a job suspends itself whenever it cannot acquire a lock to a region, deadlock conditions are frequent. A deadlock results when one job is waiting on a lock held by another job which is waiting to acquire yet another lock. In most systems job completion is a high priority. Often job completion necessitates conditions that make deadlock detection and prevention extremely complex. In stick growing, however, deadlock often means that two jobs are working on the same line. The optimal choice is to stop one of the jobs as soon as possible so that the other can complete the line.

We detect and recover from deadlock as early as possible by adding a scavenger job to the system. The scavenger job is assigned minimal priority. It runs only when there is a processor available with no outstanding search jobs. The scavenger examines the list of region locks. Whenever it finds a job suspended on a lock waiting for a job that is waiting for another region lock, it compares the priorities of the jobs. If the job holding the region has the highest priority, the scavenger does nothing. If a suspended job has a higher priority, the scavenger forces the job holding the region to roll back its search and release all locks. The scavenger job guarantees that no job is ever suspended waiting for a job with lower priority that isn't making progress.

The combination of aggressive job creation and early deadlock detection by the scavenger creates a system which make continual progress. As will be shown in the next section, our method has good performance in spite of the complex interactions of lines in most images.

4 Experimental Results

Our experimental test suite consists of 4 images ranging from very simple to very complex. Each of image measures 512x512 pixels with 256 gray scales. The test images are: `square`, `group`, `tinytown`, and `tree`. See Figure 1.

First, we conducted qualitative experiments to verify that the results produced using the parallel method where sufficiently similar to those produced by the original method. Figure 2 contains the original image, the lines extracted using the original stick growing method and lines extracted using our parallel stick growing method with 5 processors. There are some discrepancies produced by the nondeterminism of the parallel method. Parallel nondeterminism is a result of the race condition between jobs competing for the same pixels. These differences are most visible in the hexagon and triangle shapes in Figure 2. The discrepancies produced by the parallelism are no greater than those produced by the nondeterminism of the original method (e.g. when run on a shifted version of the image). The results indicate that our parallel method is qualitatively equivalent to the original method.

Our second set of experiments are quantitative in nature. We measure runtime performance of the parallel stick growing method. Runtimes were measured on our 12 processor SGI Challenge using the 20ns memory-mapped hardware clock.

Figure 3 plots the normalized execution times on 11 processors while varying the assigning priorities to search regions. Recall that throughout its lifetime a search job retains the priority assigned to the region from which it originates.

The *column major* algorithm assigns region priorities in column major order from bottom to top moving from left to right. The *row reversal* algorithm assigns regions

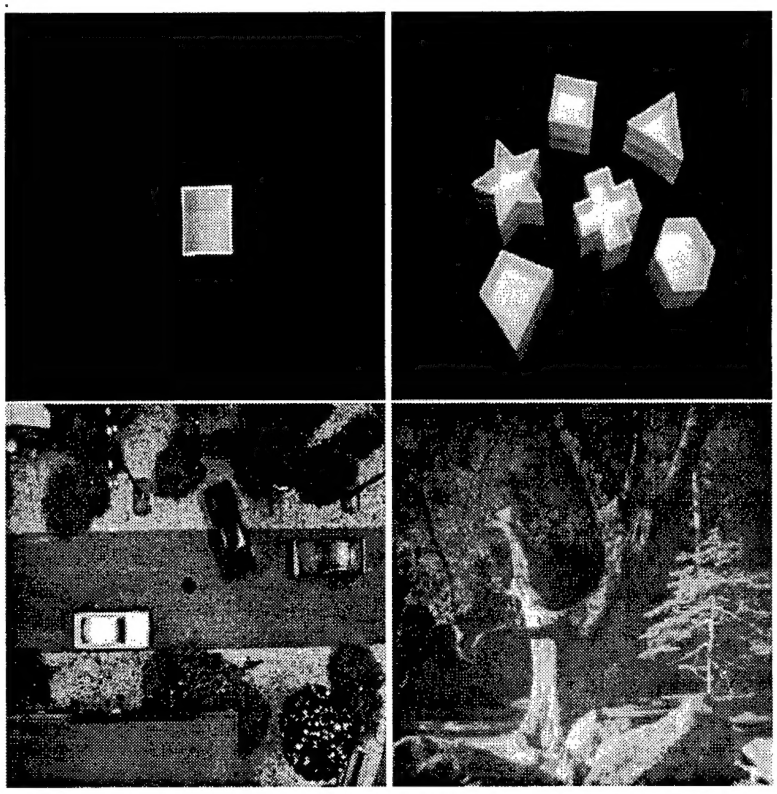

Fig. 1. Test images: `square` (TL), `group` (TR), `tinytown` (BL) and `tree` (BR).

column major priorities, but even rows move from left to right and odd rows move from right to left. The *column interleave* algorithm assigns column major priorities moving from left to right, but even columns all have higher priorities than all odd columns. The *interleave and reversal* algorithm combines the even/odd row reversal with the even/odd column interleaving. For the fifth algorithm, *random* assigns each region a unique random priority. Random priority assignment reduces the probability that that two spatially proximal searches will be active at the same time.

Figure 4 plots the speedup for each of the four sample images. Region priorities were assigned using the random algorithm. Each experiment was repeated 40 times. The average execution time pre image was calculated after discarding the first experiment to remove startup virtual memory paging latencies. Runtimes ranged from 2 seconds for 11 processors on `square` to 96 seconds for a single processor on `tree`. Appendix A contains summaries of the experimental data.

The `square` and `group` images exhibit poor scalability because their features are restricted both in number and in size. In the extreme case, the method extracts 9 lines from `square`. Since each line is be extracted by a single processor, almost all of the speed-up for simple images can be attributed to the parallelization of the preprocessing

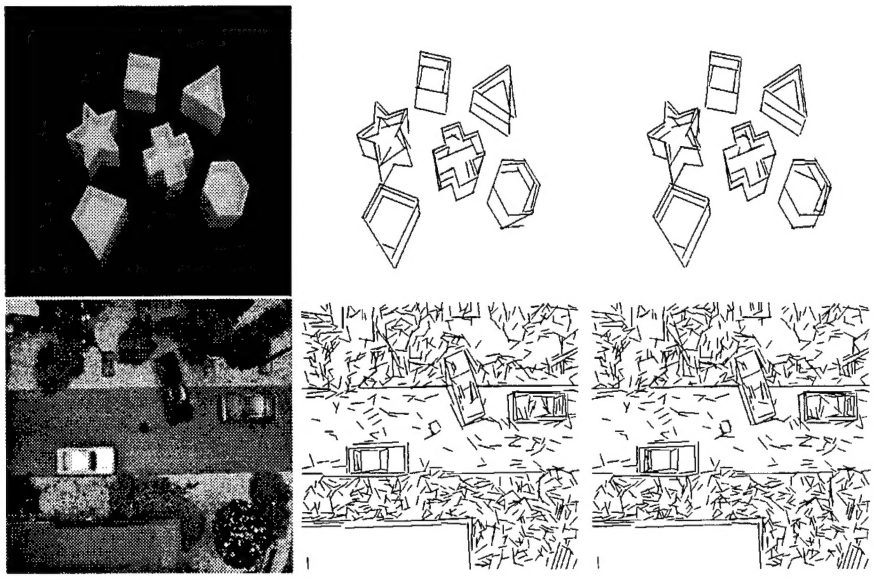

Fig. 2. `Group` (T) and `tinytown` (B): Image, Sequential Method, 5 Processor Method.

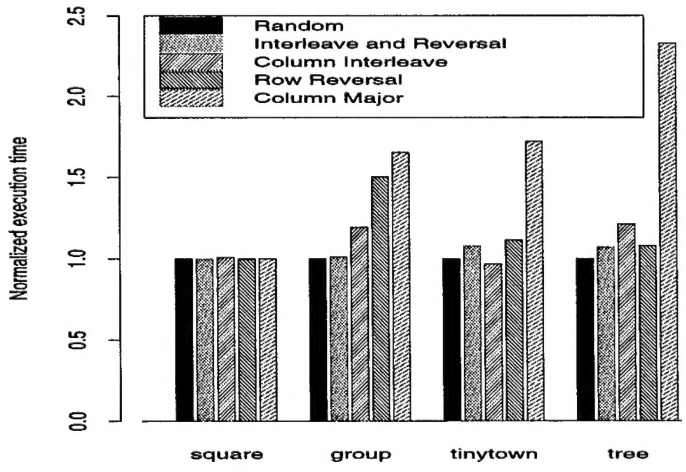

Fig. 3. Normalized execution times on 11 processors for stick growing varying the assignment algorithm.

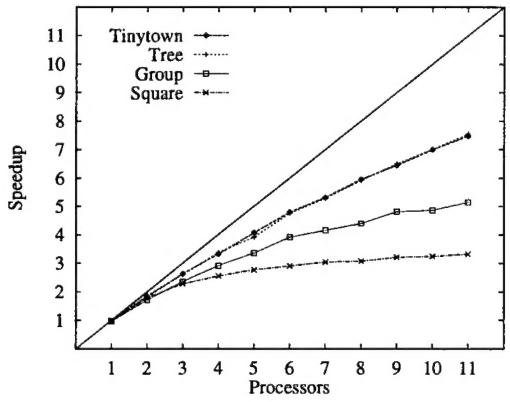

Fig. 4. Speedup of the parallel method on the sample images.

operations. As described in Section 2, the preprocessing is comprised of operations which are readily parallelized by traditional methods.

`Tinytown` and `tree` demonstrate the strong scalability of our parallel method on complex images. Since most of the processing time is spent growing line segments, two of factors limit scalability: 1) resource contention, when one or more processors attempt to acquire a lock on the same region, and 2) wasted computation, when two or more jobs attempt to grow the same line from separate regions. These effects are reduced by aggressive deadlock avoidance and priority assignment which temporally isolates spatially proximal tasks.

5 Conclusion

We have presented a parallel stick growing algorithm that is qualitatively equivalent to the sequential stick growing method. Experimental results demonstrate that our method exhibits good parallel scalability for a wide range of images. We also presented a parallel programming model using the *jobs* abstraction which is well suited to vision algorithms. The jobs abstraction provides flexible sizing in task granularity for both regular and irregular vision algorithms. A future improvement to the jobs abstraction would add a processor affinity to each job to improve cache locality. The programming model we describe is very flexible and has been ported to Sun Solaris, SGI IRIX, Digital Unix and Microsoft Windows NT. We believe our model should be readily exploitable by other vision tasks providing performance similar to that demonstrated by our parallel stick growing method. In particular, parallelizing the curvilinear variant of the sequential stick growing method should be a trivial extension of our current work.

Source Code Availability

The C++ sources for the parallelized stick-growing algorithm and the GTM package can be found at `http://www.cs.rochester.edu/u/gchunt/ipp`.

References

[1] J. B. Burns, A. R. Hanson, and E. M. Riseman. Extracting Straight Lines. In *Proc. DARPA IU Workshop*, pages 165–168, New Orleans, LA, 1984.

[2] R. O. Duda and P. E. Hart. Use of the Hough Transform to Detect Lines and Curves in Pictures. *Communications of the ACM*, 15:11–15, 1972.

[3] D. Gerogiannus and S. C. Orphanoudakis. Load balancing requirement in parallel implementations of image feature extraction tasks. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):994–1013, 1993.

[4] T. H. Hong, M. O. Shneier, R. L. Hartley, and A. Rosenfeld. Using pyramids to detect good continuation. *IEEE Transactions on Systems, Man and Cybernetics*, 13:631–635, 1983.

[5] C.-C. Lin, V. K. Prasanna, and A. Khokhar. Scalable Parallel Extraction of Linear Features on MP-2. In M. A. Bayoumi, L. S. Davis, and K. P. Valavanis, editors, *Proceedings of the IEEE Workshop on Computer Architectures for Machine Perception*, pages 352–361, New Orleans, LA, 1993.

[6] J. J. Little. Parallel Algorithms for Computer Vision on the Connection Machine. AIM-928, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1986.

[7] D. G. Lowe. *Perceptual Organization and Visual Recognition*. Kluwer Academic Publishers, Hingham, MA, 1985.

[8] A. Mansouri, A. S. Malowany, and M. D. Levine. Line Detection in Digital Pictures: A Hypothesis Prediction / Verification Paradigm. *Computer Vision, Graphics, and Image Processing*, 40:95–114, 1987.

[9] V. S. Nalwa and E. Pauchon. Edgel Aggregation and Edge Description. *Computer Vision, Graphics, and Image Processing*, 40:79–94, 1987.

[10] R. C. Nelson. Finding Line Segments by Stick Growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(5):519–523, 1994.

[11] R. Nevatia and K. R. Babu. Linear Feature Extraction and Description. *Computer Vision Graphics and Image Processing*, 13:257–269, 1980.

[12] V. K. Prasanna, C.-L. Wang, and A. Khokhar. Low Level Vision Processing on Connection Machine CM-5. In M. A. Bayoumi, L. S. Davis, and K. P. Valavanis, editors, *Proceedings of the IEEE Workshop on Computer Architectures for Machine Perception*, pages 117–126, New Orleans, LA, 1993.

[13] J. Princen, J. Illingworth, and J. Kittler. A hierarchical approach to line extraction based on the Hough transform. *Computer Vision Graphics and Image Processing*, 52:57–77, 1990.

[14] C. M. Williams. An Efficient Algorithm for the Piecewise Linear Approximation of Planar Curves. *Computer Graphics and Image Processing*, 8:286–293, 1978.

[15] Y. T. Zhou, V. Venkateswar, and R. Chellappa. Edge Detection and Linear Feature Extraction Using a 2-D Random Field Model. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 11:84–95, 1989.

A Execution Times

The following table contains maximum, average and minimum execution times for each of the test images. Also shown are the average number of line segments found in each image and the average number of aborts initiated by the scavenger thread to avoid deadlock.

square						group					
	Execution Time (secs.)						Execution Time (secs.)				
P	Max.	Avg.	Min.	Seg.	Aborts	P	Max.	Avg.	Min.	Seg.	Aborts
1	7.1267	7.0184	6.9748	9	0	1	23.5426	22.3784	21.5262	107	0
2	4.3798	3.8457	3.6324	9	7	2	14.5387	12.5891	11.2854	108	57
3	3.2556	3.0442	2.9078	9	21	3	10.6219	9.1392	7.7385	108	154
4	2.8744	2.7182	2.6418	9	26	4	8.1085	7.4060	6.2877	108	201
5	2.8162	2.5118	2.4569	9	25	5	7.5025	6.4209	5.4747	108	230
6	2.7215	2.3908	2.3309	9	26	6	6.6728	5.5070	5.2807	109	295
7	2.5799	2.2835	2.2239	9	29	7	6.2184	5.1825	4.5674	109	313
8	2.8202	2.2562	2.1422	9	29	8	5.8441	4.8979	4.4045	109	333
9	2.2166	2.1591	2.1142	9	28	9	5.4492	4.4756	3.8098	109	336
10	2.5150	2.1361	2.1035	9	28	10	5.5447	4.4283	3.7731	109	349
11	2.1337	2.0871	2.0632	9	28	11	5.2645	4.1880	3.5121	109	337

tinytown						tree					
	Execution Time (secs.)						Execution Time (secs.)				
P	Max.	Avg.	Min.	Seg.	Aborts	P	Max.	Avg.	Min.	Seg.	Aborts
1	71.2920	69.8364	67.6445	886	0	1	96.0055	94.6899	92.9571	1075	0
2	38.6185	36.8806	35.4569	886	207	2	51.9263	49.6628	48.2414	1077	387
3	26.5888	25.6757	24.7986	884	403	3	37.3478	35.5521	33.1753	1078	1 859
4	21.3605	20.4401	19.5630	885	655	4	28.8391	27.6471	26.0316	1078	11176
5	17.8572	16.6815	15.6954	887	11014	5	28.4120	23.7589	21.4923	1078	11502
6	15.3411	14.1217	12.8148	887	11158	6	20.9000	19.5797	18.3607	1079	21781
7	14.0021	12.7300	11.7680	888	11418	7	18.3213	17.5979	16.4599	1078	22024
8	12.6716	11.3598	10.2568	889	11590	8	16.4218	15.7018	14.7241	1078	22199
9	11.9200	10.5001	9.1272	889	11700	9	15.0749	14.3241	13.4812	1079	22286
10	11.0740	9.6792	8.8570	889	11767	10	13.7033	13.2774	12.6115	1080	22343
11	10.5673	9.0596	8.2577	890	11792	11	13.1274	12.3223	11.5535	1079	22390

This article was processed using the LaTeX macro package with LLNCS style